COMPILERS

Monsoon 2017 Project using the FlatB language

Dipankar Niranjan

201503003

FlatB DESCRIPTION

FlatB derives its name mainly from the fact that it has a flat namespace. There is no hierarchy of scopes. In fact, a program consists of a **declblock** followed by a **codeblock.** All the variables which will be used in the codeblock are declared in the declblock - which makes every variable a global variable.

There is only a single datatype - integers. We can have variables and one dimensional integer arrays. There can be Unary expressions and Binary expressions. Expressions can further be boolean or arithmetic.

For looping, we have the for loop and the while loop. We have the usual if-else statements. A unique feature (which in fact makes the compiler designer's life a little more painful) is the **goto** statement. FlatB supports conditional and unconditional goto(*s*).

Input is handled by read statements and output is handled by print and println statements. Each statement ends with a semi-colon. There are no functions/methods.

We've used Flex, Bison and LLVM. This means that there is a scanner.l, a parser.y, an AST.h and an AST.cpp which is the entire project really. FlatB files are files with a **.b** extension.

SYNTAX AND SEMANTICS

The CFG is as follows:

- CAPS indicate tokens.
- Since we need different types, we indicate them with a %type
- The lexer and the parser have drawn a lot from the C language's lexer and parser, respectively.
- Left associativity is followed.
- The CFG is pretty much self explanatory once one groks it.

```
%type <program> program
...

%token DECLBLOCK CODEBLOCK
...

%left '+' '-'
%left '*' '/'

program
        : decl_block code_block
        ;

decl_block
        : DECLBLOCK '{' decl_statements '}'
        ;

code_block
        : CODEBLOCK '{' code_statements '}'
        ;

decl_statements
        : declstmnt decl_statements
        |
        ;

declstmnt
        : INT declcomma ';'
        | error ';'
        ;

declcomma
        : declname ',' declcomma
        | declname
        ;

declname
        : IDENTIFIER
        | IDENTIFIER '[' NUMBER ']'
        ;

primary_arith_expression
        : IDENTIFIER
        | NUMBER
        | '(' arith_expr ')'
        | array_expr
        | '-' arith_expr
        ;
```

```
code_statements
        : codestmnt code_statements
        |
        ;

codestmnt
        : IDENTIFIER ':' genstmnt ';'
        | genstmnt ';'
        | IDENTIFIER ':' loopstmnt
        | loopstmnt
        | IDENTIFIER ':' ifstmnt
        | ifstmnt
        | error ';'
        ;

genstmnt
        : assignmentstmnt
        | printstmnt
        | readstmnt
        | gotostmnt
        ;

ifstmnt
        : IF bool_expr '{' code_statements '}' ELSE '{' code_statements '}'
        | IF bool_expr '{' code_statements '}'
        ;

gotostmnt
        : GOTO IDENTIFIER IF bool_expr
        | GOTO IDENTIFIER
        ;

loopstmnt
        : whileloop
        | forloop
        ;

whileloop
        : WHILE bool_expr '{' code_statements '}'
        ;

forloop
        : FOR array_expr '=' arith_expr ',' arith_expr ',' arith_expr '{' code_statements '}'
        | FOR IDENTIFIER '=' arith_expr ',' arith_expr ',' arith_expr '{' code_statements '}'
        | FOR array_expr '=' arith_expr ',' arith_expr '{' code_statements '}'
        | FOR IDENTIFIER '=' arith_expr ',' arith_expr '{' code_statements '}'
        ;
```

```
readstmnt
        : READ IDENTIFIER
        | READ IDENTIFIER '[' arith_expr ']'
        ;

printstmnt
        : PRINTLN arglist
        | PRINT arglist
        ;

arglist
        : STRING_LITERAL ',' arglist
        | primary_arith_expression ',' arglist
        | STRING_LITERAL
        | primary_arith_expression
        ;

assignmentstmnt
        : IDENTIFIER '=' arith_expr
        | IDENTIFIER '[' arith_expr ']' '=' arith_expr
        ;

arith_expr
        : arith_expr '+' arith_expr
        | arith_expr '-' arith_expr
        | arith_expr '*' arith_expr
        | arith_expr '/' arith_expr
        | primary_arith_expression
        ;

bool_expr
        : arith_expr '<' arith_expr
        | arith_expr '>' arith_expr
        | arith_expr LE arith_expr
        | arith_expr GE arith_expr
        | arith_expr EQ arith_expr
        | arith_expr NE arith_expr
        | '(' bool_expr ')'
        ;
```

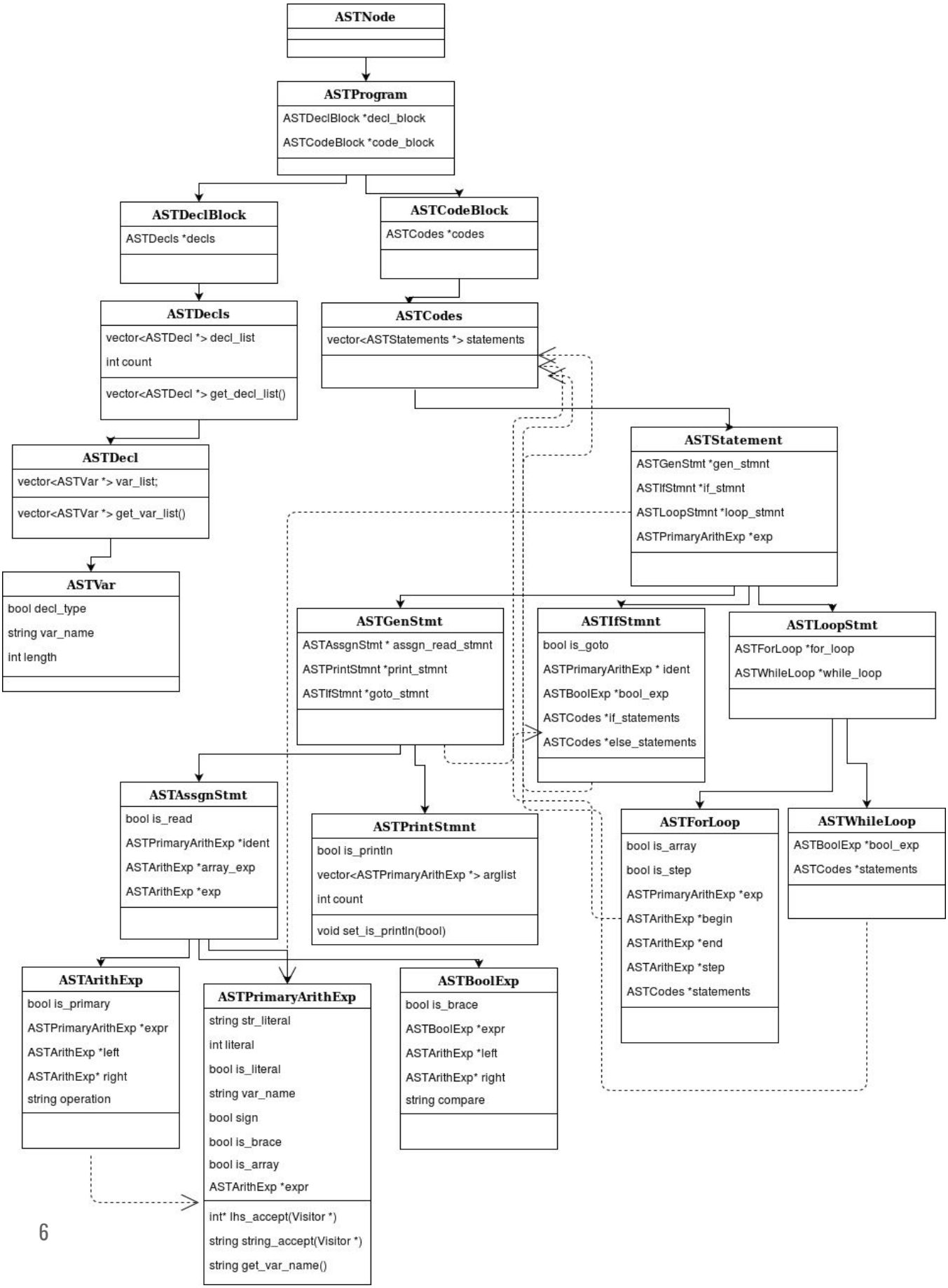
A sample .b file which covers almost everything is provided here for reference:

```
declblock{
        int i;
        int a[10];
}

codeblock{
        label1: println "first statement";
        read i;
        i = 1;
        for i=1,10{
                println "foo";
        }
        println i;
        for i=1,10{
                read a[i-1];
        }
        i = 1;
        while(i<=9){
                println a[i-1];
                i = i+1;
        }
        if(i>5){
                println "single if";
        }
        i = 6;
        if(i==5){
                println "the following is a conditional goto statment";
                goto label1 if i==5;
        }
        else{
                for i=1,5{
                        println "loop within else";
                }
        }
}
```

DESIGN OF THE AST

This diagram captures most of the details of the AST. All private variables have been shown. Only the important member methods have been shown. Constructors, Accept methods, Destructors, etc. (i.e. the run of the mill methods) have been omitted.


ASTNode
ASTProgram
ASTDeclBlock *decl_block
ASTCodeBlock *code_block
ASTDeclBlock
ASTDecls *decls
ASTCodeBlock
ASTCodes *codes
ASTDecls
vector<ASTDecl *> decl_list
int count
vector<ASTDecl *> get_decl_list()
ASTCodes
vector<ASTStatements *> statements
ASTDecl
vector<ASTVar *> var_list;
vector<ASTVar *> get_var_list()
ASTStatement
ASTGenStmt *gen_stmnt
ASTIfStmnt *if_stmnt
ASTLoopStmnt *loop_stmnt
ASTPrimaryArithExp *exp
ASTVar
bool decl_type
string var_name
int length
ASTGenStmt
ASTAssgnStmt * assgn_read_stmnt
ASTPrintStmnt *print_stmnt
ASTIfStmnt *goto_stmnt
ASTIfStmnt
bool is_goto
ASTPrimaryArithExp * ident
ASTBoolExp *bool_exp
ASTCodes *if_statements
ASTCodes *else_statements
ASTLoopStmt
ASTForLoop *for_loop
ASTWhileLoop *while_loop
ASTAssgnStmt
bool is_read
ASTPrimaryArithExp *ident
ASTArithExp *array_exp
ASTArithExp *exp
ASTPrintStmnt
bool is_println
vector<ASTPrimaryArithExp *> arglist
int count
void set_is_println(bool)
ASTForLoop
bool is_array
bool is_step
ASTPrimaryArithExp *exp
ASTArithExp *begin
ASTArithExp *end
ASTArithExp *step
ASTCodes *statements
ASTWhileLoop
ASTBoolExp *bool_exp
ASTCodes *statements
ASTArithExp
bool is_primary
ASTPrimaryArithExp *expr
ASTArithExp *left
ASTArithExp* right
string operation
ASTPrimaryArithExp
string str_literal
int literal
bool is_literal
string var_name
bool sign
bool is_brace
bool is_array
ASTArithExp *expr
int* lhs_accept(Visitor *)
string string_accept(Visitor *)
string get_var_name()
ASTBoolExp
bool is_brace
ASTBoolExp *expr
ASTArithExp *left
ASTArithExp* right
string compare

VISITOR DESIGN PATTERN

We make use of the Visitor Design Pattern for the ASTVisitor and the InterpreterVisitor code. ASTVisitor is used to construct the AST. InterpreterVisitor is used in the interpreter pass - which is **not** just another AST pass as it uses a **Program Counter** approach to handle goto statements. Both these Visitor classes inherit from a base class Visitor which contains virtual methods.

The advantage of this approach is that we can now add multiple classes which implement different specializations of the virtual functions. We don't need to modify the original class for this. Each of the visitors have accept() methods which call the respective visit() methods.

The CodeGenVisitor is also created, but this does not strictly adhere to the pattern as it doesn't make use of the accept() methods. The CodeGen phase consists of CodeGenVisitor::visit() methods for each of the ASTNode's derived classes which call each other directly, instead of through accept() methods.

A five minute examination of the code is sufficient to understand how this design pattern has been used and how it makes our life easier.

DESIGN OF THE INTERPRETER

The Interpreter is not just one more AST pass. Since the goto statement is a part of the language, execute recursively" doesn't fit well in this scenario. For goto to work, we need some kind of a PC, a "program counter" and each statement in the program must have a distinct address. In this case, since we maintain a vector of statements, the PC is nothing but the index variable of the vector. When goto is encountered, the target index variable of the goto (it's argument) is put into the PC and the execution resumes from there (a previous pass stores indexes of various labels)

This is almost impossible to achieve with a stack-based, recursive approach. We'll have to resort to hacks to accomplish that. Something like when goto is encountered, throw a GotoException which breaks out of all of the stack frames and goes back to the root. We process statements (skip them without executing) until we reach the target address - basically very slow.

Storage for the variables in the declblock is allocated and all operations on these variables are reflected in the storage. We make use of the InterpreterVisitor to execute code from the AST.

DESIGN OF LLVM CODE GENERATOR

Yet again, something similar to the Visitor Design Pattern has been used - CodeGenVisitor(s) - which accept objects of ASTNode(s) and emit LLVM IR to stdout. Though FlatB doesn't have functions/methods, we've created a main method (which is shown in the emitted IR) so that the entry point into the program is unambiguous.

We maintain a symbol table which keeps track of the declared variables (in our case everything is a global variable) and also of the basic blocks. We appropriately change/create basic blocks as per the control flow (which is known while traversing the AST) We use the verifyModule() for verification.

Each of the CodeGenVisitor(s) returns a void * pointer or nothing. If a void * pointer is returned, it can be casted appropriately into the required types (llvm::Value * or others) The main challenge is to handle creation and control flow of basic blocks in the if/else statements and the loop statements. Appropriate BranchInst have to be created and inserted. Having a single variable type - integer - simplifies a lot of things.

PERFORMANCE COMPARISON

CPU CLOCK (msec) and INSTRUCTIONS

Interpreter

BUBBLE SORT	500	1000	1500	2000
For loop	**80.63** 70,83,13,275	**363.32** 3,54,92,69,542	**982.23** 9,65,86,10,784	**2044.01** 20,16,18,98,678
While loop	**99.78** 87,52,52,435	**439.67** 4,21,06,31,647	**1155.06** 11,14,33,37,009	**2341.03** 22,79,89,98,450

lli

BUBBLE SORT	500	1000	1500	2000
For loop	**14.37** 3,81,77,766	**17.03** 4,55,90,593	**16.65** 5,79,79,249	**17.47** 7,50,02,803
While loop	**12.43** 3,81,97,301	**12.80** 4,57,95,698	**17.10** 5,80,94,022	**16.95** 7,52,40,366

llc

BUBBLE SORT	500	1000	1500	2000
For loop	**15.62** 3,54,38,263	**15.17** 3,54,18,641	**10.28** 3,55,04,262	**14.22** 3,55,68,994
While loop	**10.96** 3,55,60,255	**15.41** 3,56,60,563	**10.93** 3,56,58,581	**17.55** 3,57,71,291

Interpreter

ARRAYMUL	10000	100000
No passes	**75.79** 71,62,53,030	**5933** 61,02,34,40,265

Command for Optimization Passes: opt -adce -bb-vectorize -constprop -die -globalopt -instcombine -stats ircode.ll -S -o ircodedup.ll

lli

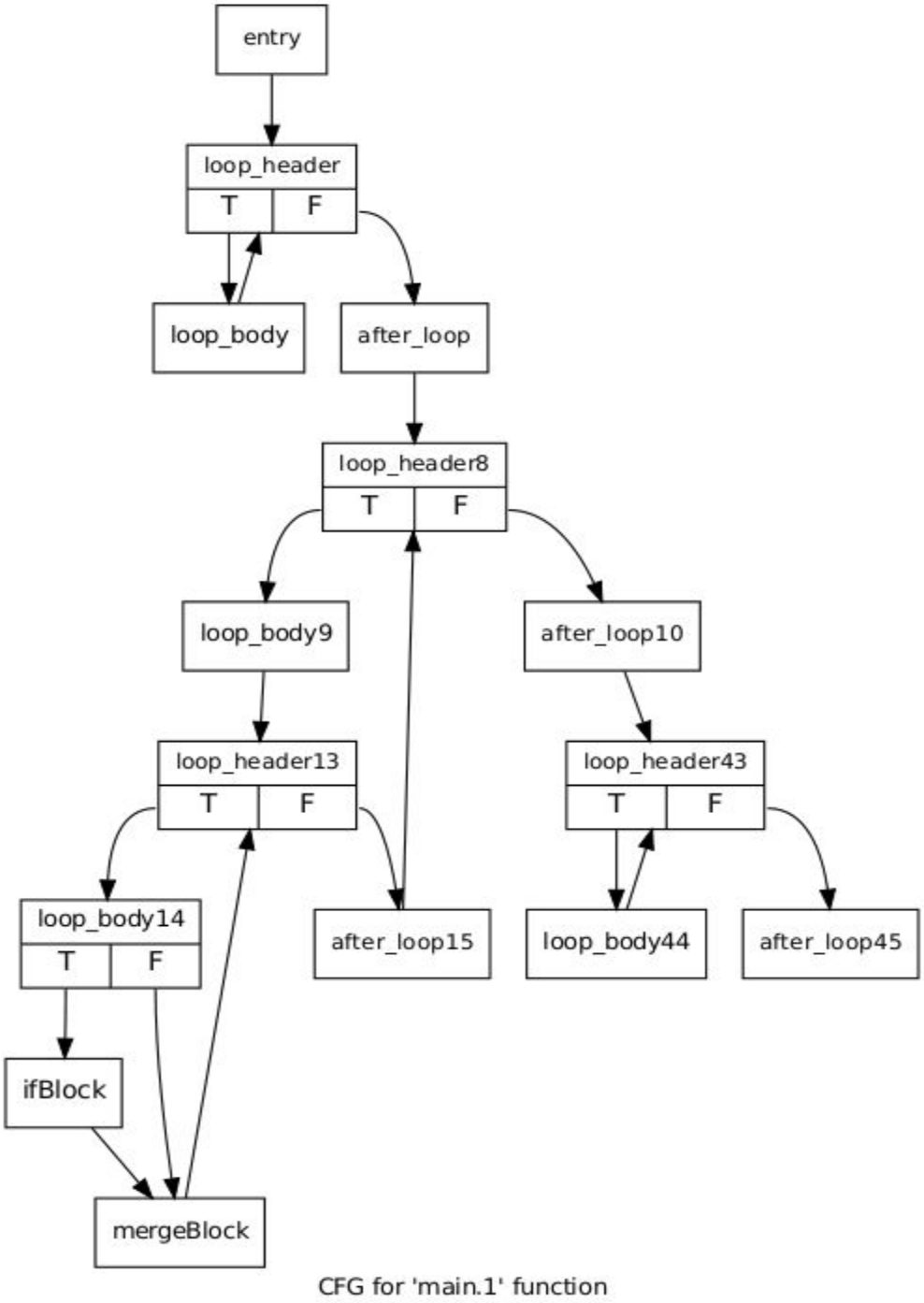
ARRAYMUL	10000	100000
No passes	**14.72** 5,02,29,663	**31.48** 15,55,94,856
Passes	**15.89** 5,07,11,051	**31.97** 15,66,93,915

llc

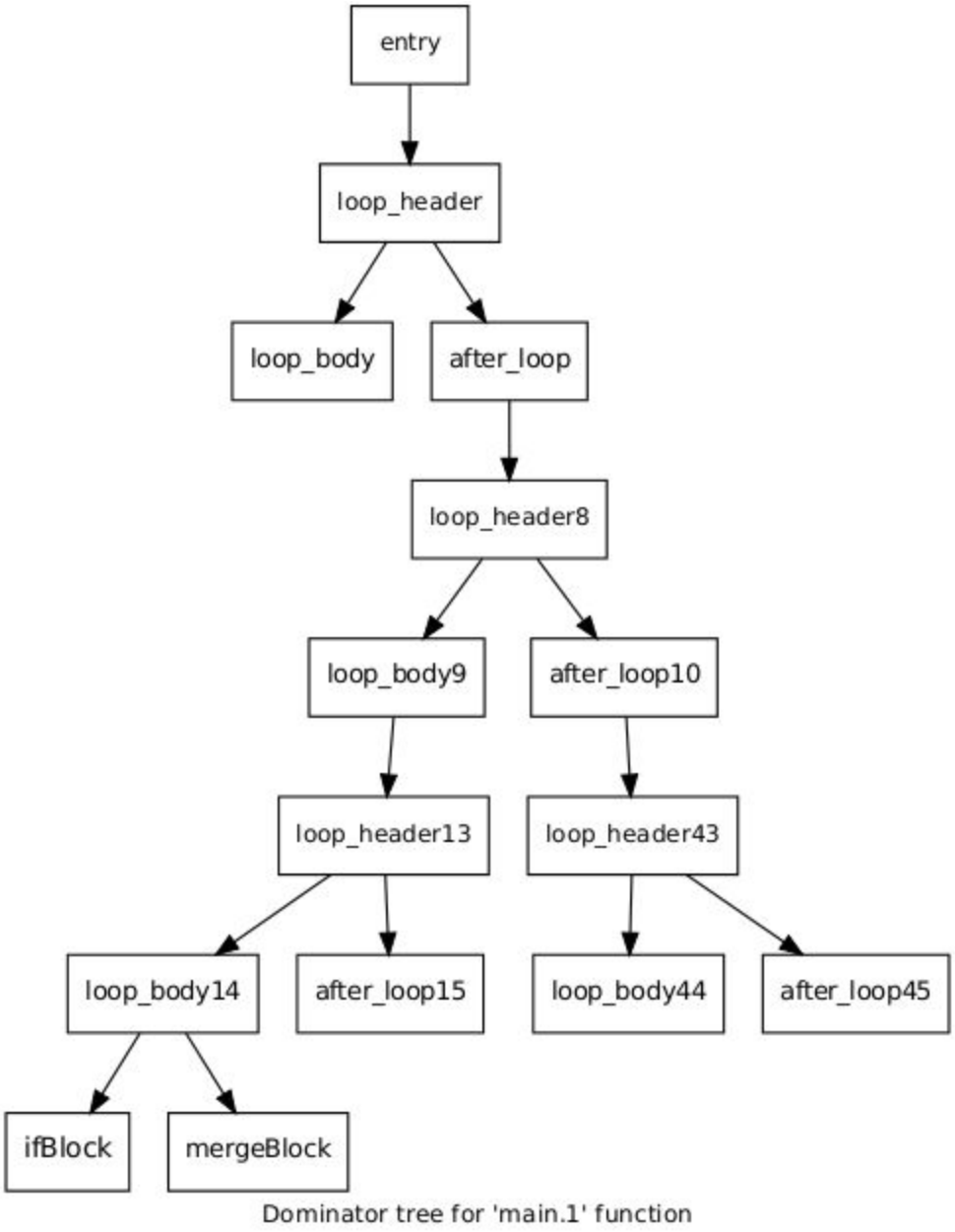
ARRAYMUL	10000	100000
No passes	**13.63** 3,87,94,708	**15.81** 3,87,94,585
Passes	**13.25** 3,91,17,109	**12.83** 3,90,73,969

BINARY SEARCH	100000	1000000
Interpreter	**1502.81** 15,26,98,33,437	-
lli	**22.85** 12,54,69,347	**119.84** 96,97,99,655
llc	**15.81** 3,62,36,496	**15.59** 3,62,28,508

BONUS



CFG for ‘for’ bubblesort



Dominator Tree for ‘for’ bubblesort